Toward A Common Framework for Statistical Analysis and Development

Toward A Common Framework for Statistical Analysis and Development

Kosuke IMAI, Gary KING, and Olivia LAU

We develop a general ontology of statistical methods and use it to propose a common framework for statistical analysis and software development built on and within the R language, including R's numerous existing packages. This framework offers a simple unified structure and syntax that can encompass a large fraction of existing statistical procedures. We conjecture that it can be used to encompass and present simply a vast majority of existing statistical methods, without requiring changes in existing approaches, and regardless of the theory of inference on which they are based, notation with which they were developed, and programming syntax with which they have been implemented. This development enabled us, and should enable others, to design statistical software with a single, simple, and unified user interface that helps overcome the conflicting notation, syntax, jargon, and statistical methods existing across the methods subfields of numerous academic disciplines. The approach also enables one to build a graphical user interface that automatically includes any method encompassed within the framework. We hope that the result of this line of research will greatly reduce the time from the creation of a new statistical innovation to its widespread use by applied researchers whether or not they use or program in R.

Key Words: Graphical user interface; Interdisciplinary; R language; Statistical ontology; Statistical software.

1. INTRODUCTION

Quantitative methodology is thriving like never before, both in the discipline of statistics and in the quantitative subfields of diverse substantive disciplines. Despite a common underlying mathematical and statistical foundation and numerous cross-disciplinary

Kosuke Imai is Assistant Professor, Department of Politics, Princeton University (Corwin Hall, Department of Politics, Princeton University, Princeton NJ 08544 (E-mail and Web: *KImai@Princeton.Edu* and *http://Imai.Princeton.Edu*. Gary King is David Florence Professor of Government, Harvard University (Institute for Quantitative Social Science, 1737 Cambridge Street, Harvard University, Cambridge MA 02138 (E-mail and Web: *King@Harvard.Edu* and *http://GKing.Harvard.Edu*). Olivia Lau is Ph.D. Candidate, Department of Government, Harvard University (1737 Cambridge Street, Cambridge MA 02138 (E-mail and Web: *Olivia.Lau@Post.Harvard.Edu* and *http://www.OliviaLau.Org*).

Journal of Computational and Graphical Statistics, Volume 17, Number 4, Pages 1–22
DOI: 10.1198/106186008X384898

efforts, however, quantitative analysis looks remarkably different in each substantive discipline. Researchers in different fields favor different jargon, different mathematical notation, different parameterizations, different quantities of interest, and different syntax for computer implementation. Traversing the quantitative methods subfields of these diverse disciplines and understanding what the natives have to offer—in everything from statistical theory to software implementation—can be highly productive, but has often been far more difficult than it should be for pursuits that have so much underlying structure in common.

Among the efforts to reduce the costs of spanning these diverse subfields, the R Project for Statistical Computing (Ihaka and Gentleman 1996; R Development Core Team 2008) and the S language on which it is based (Becker, Chambers, and Wilks 1988) stand as monumental developments. These projects solve so many problems for developers that a large fraction of statistical innovators from many fields now implement their methods first as R-language programs and distribute them as open source R packages. This development makes it possible for statistically sophisticated researchers who are literate in programming languages to use new methods soon after their creation.

Unfortunately, using R packages is not always easy even for sophisticated users, given the diverse syntax, far-flung examples, and uneven documentation quality. Those who are statistically sophisticated but do not know how to program will have more difficulty using the new procedures. Researchers who do not use R must wait for other (mostly commercial) statistical packages to reprogram the new procedures from scratch. And of course, since the vast majority of applied data analysts do not know R, and are unlikely to use statistical software without an easy-to-use graphical user interface (GUI) during their entire careers, the time from statistical innovation to widespread use is still far too long.

We propose to make progress on this problem in a way that does not require statistical innovators to change existing practices. Instead, developers can supplement what they do now with a few simple bridge functions that translate their chosen approach into a common framework. Using a new method of wrapping and then extending existing packages, our framework consists of three steps—fitting a statistical model, choosing a quantity of interest by specifying the values of explanatory variables, and implementing simulation to make inferences about predetermined quantities of interest (although users can also compute other arbitrary quantities of interest from the output)—which gives applied users a relatively universal statistical syntax with three general-purpose commands to perform the procedure for any included R package. The bridge functions we recommend tap into our ontology for describing statistical models so that GUIs can be created automatically, without additional programming to include new packages. These developments should make it easier for R developers to reach a significantly larger audience without much additional effort.

The result of this work is not only a simple user interface. It also enables developers and users to take advantage of infrastructure that works for a wide range of methods without having to build it themselves. For example, we introduce here an intuitive extension of R's existing single equation formula framework to encompass a much larger range of statistical models (and model-free approaches) so they can work with multiple equation, multilevel, hierarchical, panel, time series, and other structures, which reduce the complex-

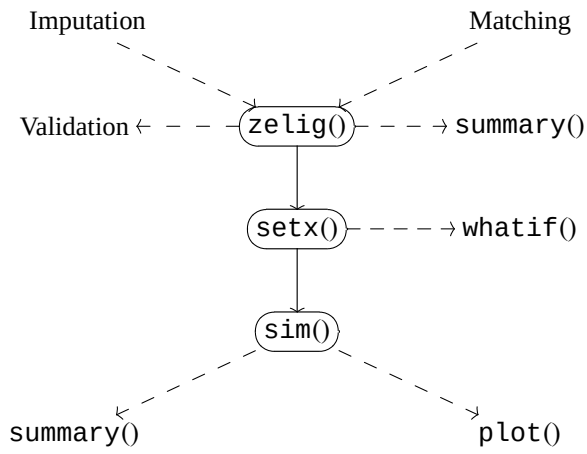

Figure 1. Main Zelig commands (solid arrows) and some options (dashed arrows).

ities of managing multiple datasets, groups of covariates, and parameter vectors in these settings. Among other innovations, we have also added facilities for creating and rerunning replication datasets, using multiply imputed data and data processed via matching, running multiple analyses within given strata, bootstrapping parameters, documenting statistical models through a standard framework, and translating often uninterpretable model parameters into quantities of direct scientific interest.

We describe these developments first from the perspective of the user (Section 2) and then from that of the developer (Section 3). We offer an implementation of these ideas, and illustrate them throughout this article, via an R package called Zelig (Imai, King, and Lau 2006). However, all the ideas we describe here exist independently of our software and can be adopted or extended separately from our particular implementation. This article summarizes only the structural aspects of Zelig, rather than all of its options. See the Zelig project Web site, at *http://gking.harvard.edu/zelig*, for more information. (We named Zelig after a Woody Allen movie about a man who had the strange ability to become the physical and psychological reflection of anyone he met and thus to fit perfectly in any situation).

2. A UNIFIED USER INTERFACE

From the user's perspective, we organize data analysis into three core activities which are a part of Zelig, and a variety of related activities. Figure 1 outlines the main features. The basic idea is that raw data goes in—perhaps after being preprocessed via matching methods for causal inference (Rubin 1973; Ho et al. 2007), multiple imputation for missing data (Rubin 1987; King, Honaker, Joseph, and Scheve 2001; Honaker and King 2007), or outlier removal and feature detection to improve data quality or statistical robustness (Bishop 1995, chap. 8)—and then three commands are always performed: First, some statistical method, such as a likelihood or Bayesian model, is specified and fit. Second, we

identify the quantity of interest, such as forecasts, causal effects, or conditional or unconditional counterfactual evaluation. This is usually done by setting each of the explanatory variables to one or more chosen (actual or counterfactual) values. Finally, the quantity of interest can be computed by simulation drawn using an asymptotic normal approximation, bootstrap resampling, a Bayesian posterior simulation, or any other available method. The corresponding three commands to implement these three stages in Zelig are `zelig()`, `setx()`, and `sim()`, respectively. In one simple example,

```
z.out <- zelig(y ~ age + race, model = "logit",
         data = turnout)
x.out <- setx(z.out, age = 36, race = "white")
s.out <- sim(z.out, x = x.out)
```

All code selections displayed in this article, including the above, are executable demos included in the Zelig package, except where clearly identified as pseudo-code. Since the code above requires some set up, such as commands to load and recode data, we provide only excerpts in the interests of parsimony. To view the full example above, use `demo(logit)` after loading the Zelig library within R.

As input, Zelig accepts R data frames, or preprocessed data output from the R packages Amelia II (Honaker, King, and Blackwell 2006) for imputing missing data and MatchIt (Ho et al. forthcoming) for performing matching to reduce model dependence for causal inference.

Output from each of these three steps may be evaluated or viewed. For example, the output of `zelig()` can be summarized using existing methods for goodness of fit, residual analysis, and the like. Since quantities of interest farther from the data are more model dependent, we can evaluate the output of `setx()` to via the R package `whatif` to determine how far the counterfactual question of interest is from the data (see King and Zeng 2006; King and Zeng 2007; Stoll, King, and Zeng 2005). One may also use diagnostic tools such as cross-validation to validate the fitted model for any model supported by the Zelig framework. Finally, estimates of the quantities of interest may be studied in any desired format, including point estimates and standard errors, confidence (or credible) intervals, likelihood functions, or posterior densities.

We now turn to our specific innovations in interpreting and presenting statistical results (Section 2.1), and generalizing R formulas (Section 2.3).

2.1 INTERPRETING AND PRESENTING STATISTICAL RESULTS

From a user's perspective, one of the most confusing aspects of learning a statistical procedure comes after the arduous steps of acquiring, cleaning, recoding, describing, and exploring the data, choosing a statistical model, getting the data into the computer program, coding up the particular model you want to run or figuring out the syntax of a pre-existing package: the statistical results typically come out of the program on the scale convenient to the programmer, rather than the user. Model parameters in logit, probit, and negative binomial regressions as well as hundreds of other procedures all need to be interpreted

differently; few are documented; and only rarely are model parameters on a scale that have any direct scientific meaning. Scholars feel the need to present tables of model parameters in academic articles (perhaps just as evidence that they ran the analysis they claimed to have run), but these tables are rarely interpreted other than for their sign and statistical significance. Most of the numbers in these tables are never even discussed in the text. From the perspective of the applied data analyst, R packages without procedures to compute quantities of scientific interest are woefully incomplete.

A better approach focuses on quantities of direct scientific interest rather than uninterpretable model parameters. The lists of parameter estimates and standard errors are then treated as intermediate values, used to calculate quantities of interest that are typically on the scale of the variable being predicted or explained. For example, the distributions of predicted values and expected values obtained from the model may be compared to the empirical distribution of the dependent variable observed in the data. These quantities include counterfactual predictions, or what the value of the dependent variable would have been if the explanatory variables had taken on particular values. They include causal effects, which are normally taken to be some type of comparison between the observed value of the outcome variable and a counterfactual, such as if a treatment were applied to a control unit. The comparison in causal effects can be done by ratios, differences, or other calculations, and produces quantities such as relative risks, risk differences, attributable risks, first differences, marginal effects, average treatment effects, average treatment effects on the treated, numbers needed to treat, and so forth.

For each quantity of interest, the user needs some summary that includes a point estimate and a measure of uncertainty such as a standard error, confidence interval, or a distribution. The methods of calculating these differ greatly across theories of inference and methods of analysis. However, from the user's perspective, the result is almost always the same: the point estimate and uncertainty of some quantity of interest. To calculate the quantities of interest, both the ones we choose *ex ante* and others that developers may choose themselves, we use the fact that for almost every statistical procedure there exists a method of simulating parameters, and that any quantity of interest can be computed from these simulations. By replicating several published articles and computing simulations of several quantities of interest, King, Tomz, and Wittenberg (2000) showed that following this procedure can produce considerable information of direct interest to researchers, information which is not readily available through the usual presentation of coefficients.

This procedure consists of three steps, which correspond to the three functions in Figure 1. First, fit a statistical model via the `zelig()` function, which wraps many existing statistical procedures. At this point, users may choose to use sets of multiply imputed data frames, matched data, or subsets of the data by running the analysis separately in each stratum (see Section 2.2). The output of `zelig()` also can be used for model validation using, for example, a cross-validation procedure.

Second, select quantities of interest you would like to compute or calculate by calling `setx()`. Choose values of the explanatory variables for a forecast, or use two calls to `setx()` to compute a causal effect. For example, we might compute a hypothetical causal effect by setting the treatment variable at 1 and then 0, while holding the other explanatory

variables constant (whether at their means, modes, or medians). Alternatively, compute the in-sample average treatment effect (Imbens 2004) by setting the explanatory variables at their observed values, and imputing only the unobserved counterfactual for each individual.

Finally, use the model output from `zelig()` and the values for the explanatory variables from `setx()` to compute simulations of the quantities of interest using the `sim()` command. This procedure involves simulating parameters from their sampling or posterior distributions, or the (conceptual) equivalent in other theories of interest, using the simulated parameters to compute simulations of the dependent variable, and then calculating any quantity of interest from these simulations.

The simulations are then viewed via generic `summary()` or `plot()` commands to report the list of precoded quantities of interest (given the choice of values of explanatory variables). The raw simulation draws of course remain available so that new quantities of interest can always be computed by the user (e.g., the probability that income is less than the poverty level).

2.2 INFRASTRUCTURE FOR REPLICATION DATASETS, BOOTSTRAPPING, MULTIPLE IMPUTATION, AND MULTIPLE ANALYSES

By recognizing the common features of many statistical models, it becomes possible to add infrastructure that would benefit anyone using models implemented in the same framework. In Zelig, we have simple options to bootstrap parameters, run multiple analyses within strata specified by the user, use lists of datasets to deal with multiply imputed data, or employ combinations of the three options. From the `mi` demo in Zelig, we can run analyses for multiply imputed datasets using

```
z.out <- zelig(as.factor(ipip) ~ wage1992, model = "ologit",
               data = mi(immi1, immi2, immi3, immi4, immi5),
               by = "gender")
x.out <- setx(z.out)
s.out <- sim(z.out, x = x.out)
```

Any other procedure that can be applied uniformly to the broad class of statistical models covered here can easily be included and applied to any individual model.

With this framework, users can also create and store "replication datasets," which many scholarly journals now archive (King 1995). Although replication datasets can be as crude as a zip file with a read-me including a narrative of what was done and some raw data, R provides users with the option of storing data, output, quantities of interest, and the code used to generate them in a single R image file. For example, if `z.out` is the `zelig()` output, and `s.out` is the `sim()` output, we can create and save the replication data file as in the `repl` demo:

```
save(turnout, z.out, s.out, file = "demo_replication.RData")
```

Zelig provides a replication procedure that reduces the many steps involved in data analysis into one command. After loading the replication data file, the generic function `repl()`

will either reevaluate the model, or in the case of output from `sim()`, reevaluate the model and recalculate quantities of interest. Continuing the `repl` demo:

```
load("demo_replication.RData")
s.rep <- repl(s.out)
```

If the random seed was saved with the replication materials, the replication object `s.rep` will return identical quantities of interest.

2.3 GENERALIZING R FORMULAS

The base R framework includes a successful and widely adopted "formula" framework for identifying, transforming, including, and excluding dependent and explanatory variables for single equation models. (The syntax of the basic version is an outcome variable, a tilde separator, and a list of explanatory variables separated by plus signs: `y ~ x1 + x2`, where "+" means inclusion, not addition.) This provides an easy-to-use syntax, but its most important contribution is recognizing and taking advantage of the fact that many models have common features: a dependent variable and one set of associated explanatory variables. The features that vary across models, such as the functional form and stochastic component, are identified separately from the variables selected.

We now expand on this insight and identify common features of a much broader class of statistical models, including single equation, multiple equation, time series, multilevel, and hierarchical models, as well as those with constraints across equations. Identifying these common features lets us write a simple generalization of the R single equation formula that applies much more widely. This may also simplify notation across packages: Although the same R single equation formula is used in a wide array of packages, procedures that take more than one set of explanatory variables now use almost as many different syntaxes as there are packages.

We begin by recognizing that a large class of statistical models all have *stochastic* and *systematic* components. The stochastic component specifies a scalar or vector of dependent variables Y_i (for observations $i = 1, \ldots, n$) distributed as $P(\mu_i, \theta)$, where P is a density that may or may not be known. The systematic components involve parameters μ_i that vary over the observations and parameters θ that are constant over observations. Each of the elements of μ_i varies as a (known or unknown) function $g(\cdot)$ of (measured or latent) explanatory variables X_i and fixed parameters, β, such that $\mu_i = g(X_i, \beta)$. The model is completed with some independence assumption, most typically that Y_i and Y_j are independent conditional on μ_i and θ for all $i \neq j$ (see King 1989). For example, the simple linear-normal regression model has a scalar dependent variable Y_i distributed normally with mean μ_i and variance σ^2, and with the mean varying as a linear function of a vector of covariates X_i, and a conformable vector of coefficients, β:

$$Y_i \sim N(\mu_i, \sigma^2), \quad \text{where} \quad \mu_i = X_i\beta. \tag{2.1}$$

All statistical procedures in this broad class have four key features. First, models may have more than one systematic component, such as if we add a variance component to (2.1):

$\sigma_i^2 = \exp(Z_i\gamma)$, with a vector of explanatory variables Z_i (that may overlap with X_i) and a parameter vector γ. Second, the parameters in these equations may have constraints that require coefficients on some variables in different equations to be equal. Third, equations in statistical models represent the parameters of distributions, such as σ_i^2 above, and do not necessarily correspond to specific dependent variables. Finally, for some multilevel or hierarchical models, different parameters may be logically associated with variables from different datasets.

To ensure that our generalized formula framework can incorporate these ideas, we first provide a way to identify (and thus to constrain) coefficients. For example, the equation `y ~ x1 + tag(x2,"beta") + x3` labels (or "tags") the coefficient on `x2` as `beta`. We also allow a list of equations. The combination of the two enables one to specify constraints across equations. For example the list of formulas `list(y~ tag(x1,"gamma") + x2, y2 ~ z1 + tag(z2,"gamma"))`, constrains the coefficient on `x1` in the first equation to equal the coefficient on `z2` in the second (since both have the same label `gamma`).

We also need a way to label parameters separately from the dependent variables to which they may correspond. We do this by allowing each formula in the list to be labeled with a name corresponding to the parameter it represents, via the standard method for labeling elements in lists. For example, we can represent the normal model with variance function as `list(mu = y ~ x1 + x2, sigma = ~ z1 + z2)`, where the equation for the variance component has no corresponding dependent variable on the left-hand side.

Since this particular model always includes exactly two equations, only one of which has a dependent variable, dropping the names in the list would not create ambiguity and so is allowed. We could also represent the model defined in Equation (2.1), that is without the variance component, more completely as `list(mu = y ~ x1 + x2, sigma = ~ 1)`, but as a default we would not suggest requiring elements that are unnecessary, and so this expression reduces to the current R standard formula, `y ~ x1 + x2`.

We now illustrate how this generalized formula framework can represent four more sophisticated models and, where necessary, we also introduce other features of our framework.

A Bivariate Probit Model. The bivariate probit model has dependent variables $Y_i = (Y_{i1}, Y_{i2})$ observed as (0,0), (1,0), (0,1), or (1,1) for all i. The stochastic component with two latent bivariate normal variables (Y_{i1}^*, Y_{i2}^*) is

$$\begin{pmatrix} Y_{i1}^* \\ Y_{i2}^* \end{pmatrix} \sim \mathrm{N}\left\{ \begin{pmatrix} \mu_{i1} \\ \mu_{i2} \end{pmatrix}, \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \right\}, \tag{2.2}$$

with marginal means $\mu_{i1} \equiv E(Y_{i1}^*)$ and $\mu_{i1} \equiv E(Y_{i1}^*)$ and correlation $\rho \equiv \mathrm{cor}(Y_{i1}^*, Y_{i2}^*)$. The following observation mechanism links the observed dependent variables, Y_{ij}, with the latent variables

$$Y_{ij} = \begin{cases} 1 & \text{if } Y_{ij}^* \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

for $j = 1, 2$. The model has three systemic components, each with separate but possibly overlapping vectors of explanatory variables, X_i, Z_i, and W_i, respectively:

$$\mu_{i1} = X_i\beta, \quad \mu_{i2} = Z_i\gamma, \quad \text{and} \quad \rho_i = \frac{\exp(W_i\theta) - 1}{\exp(W_i\theta) + 1}. \tag{2.4}$$

We can represent this model in three formulas with a constraint across two of them for illustrative purposes:

```
formulae <- list(mu1 = y1 ~ tag(x1,"beta") + x2,
                 mu2 = y2 ~ tag(z1,"beta") + z2,
                 rho =    ~ w1 + w2)
```

where, by design, no dependent variable is associated with the `rho` equation.

A Compound Hierarchical Ordered Probit Model. Models with parameters fitted from different datasets also fit our framework. This model corrects survey responses due to threshold shifts resulting from differential item functioning (i.e., survey respondents having different standards for what constitutes different levels of the dependent variable; see King, Murray, Salomon, and Tandon 2004). The main portion of the model is a multivariate ordered probit, for independent normal latent variables Y_{is}^* for observation i ($i = 1, \ldots, n$) and self-assessment variable s ($s = 1, \ldots, S$). The stochastic components for the latent variables are normal, $Y_{is}^* \sim N(\mu_i, 1)$ for all s, with a common systematic component, $\mu_i = X_i\beta$. As in ordered probit, for each equation s, we only observe y_{is}, which indicates the category into which the latent variable Y_{is}^* falls:

$$Y_{is} = k \qquad \text{if } \tau_{is}^{k-1} \le Y_{is}^* < \tau_{is}^k \tag{2.5}$$

with a vector of thresholds τ_{is} (where $\tau_{is}^0 = -\infty$, $\tau_{is}^{K_s} = \infty$, and $\tau_{is}^{k-1} < \tau_{is}^k$, with indices for categories $k = 1, \ldots, K_s$ and self-assessment questions $s = 1, \ldots, S$) that vary over the observations as a function of a vector of covariates, V_i (which may overlap X_i), and a vector of unknown parameter vectors, γ_s, with elements the vector γ_s^k:

$$\tau_{is}^1 = \gamma_s^1 V_i \tag{2.6}$$

$$\tau_{is}^k = \tau_{is}^{k-1} + e^{\gamma_s^k V_i} \qquad (k = 2, \ldots, K_s - 1). \tag{2.7}$$

Finally, there exists a set of latent positions on vignette questions $Z_{\ell j}^*$ possibly from a different survey (with observations $\ell = 1, \ldots, L$) with constant means:

$$Z_{\ell j}^* \sim N(\theta_j, \sigma_j^2), \tag{2.8}$$

which are turned by the respondent into a categorical answer to the survey question $z_{\ell j}$ via the observation mechanism:

$$z_{\ell j} = k \quad \text{if} \quad \tau_{\ell 1}^{k-1} \le Z_{\ell j}^* < \tau_{\ell 1}^k \tag{2.9}$$

and with thresholds determined by the same γ_1 coefficients as in (2.6) for Y_{i1}, and the same explanatory variables V_ℓ, with values measured for the second dataset with indices labeled ℓ, V_ℓ:

$$\tau_{\ell 1}^1 = \gamma_1^1 V_\ell \tag{2.10}$$

$$\tau_{\ell 1}^k = \tau_{\ell 1}^{k-1} + e^{\gamma_1^k V_\ell} \qquad (k = 2, \ldots, K_1 - 1).$$

This more complicated model is easily represented with the tools above, since the same common features also apply to this model. We need one equation for each μ corresponding to a self-assessment question, one for each θ corresponding to a vignette question from a possibly different dataset; and one equation for τ_i and one for τ_ℓ, corresponding to no dependent variable, but possibly different datasets. Following the `chopit` demo:

```
formulas <- list(self = y ~ sex + age + factor(country),
                 vign = cbind(v1, v2, v3, v4, v5) ~ 1,
                 tau  = ~ sex + age + factor(country))
```

where the `vign` equation is shorthand for `vign1 = v1 ~ 1, ..., vign5 = v5 ~ 1`, which is possible because the right side of both equations are always identical in this model (as they are estimated with scalar mean, and hence do not take explanatory variables). Since each equation has a label, we can load variables in each equation from a different data set. For example, continuing the `chopit` demo,

```
data <- list(self = free1, vign = free2)
z.out <- zelig(formulas, data = data, model = "chopit")
```

Since τ is drawn from both datasets, this model does not require it to be explicitly identified in the data statement.

Time Series Models. We implement a user-interface for single equation time series models with special functions for differencing, `Diff(Y,d)`; lags of Y for AR(p) terms, `lag.y(p)`; and lags of the disturbance for MA(q) terms, `lag.eps(q)`. For example, we can represent an ARIMA(3,1,2) model without a covariate as $y_t \sim \mathrm{N}(\mu, \sigma)$, where

$$E(y_t) = \mu_t = \sum_{j=1}^{3} \beta_j y_{t-j} + \sum_{j=1}^{2} \gamma_j \epsilon_{t-j} \tag{2.11}$$

and where $y_t = Y_t - Y_{t-1}$ and $\epsilon_t = y_t - E(y_t)$ via the intuitive representation, `Diff(y,1) ~ lag.y(3) + lag.eps(2)`. To view this example with and without covariates, see the `arima` demo in Zelig.

Multilevel Models. Multilevel models are also easy to represent in this framework by tagging a coefficient on an explanatory variable in one equation and using the tag as the name (for the list element) of another equation. For this use, we introduce a `|` (which often means "by" in R) and a group identification (or strata) variable to be included in the `tag()` special. For a simple example, we could have `list(y ~ x1 + tag(x2, gamma | state), gamma = ~ z1 + z2)`, where the first equation varies over all individuals and the second varies over the aggregate `state` variable. The label, rather than an explicit parameter name, identifies the second equation.

Our framework also allows both structural and reduced forms of equations. We usually regard the structural version, such as in the previous paragraph, as most intuitive, but the reduced form is sometimes more convenient. For example, the same model can be represented in reduced form as `list(y ~ x1 + tag(x2, z1 + z2 | state))`. In addition, if the user chooses to have all variables in one dataset, then all variables from both

equations would be recorded at the individual level and only one dataset would specified in the data argument in `zelig()`. With the tools offered here, considerably more elaborate models can be included in the same framework.

As an example, consider a multilevel logistic regression model for individual $i = 1, \ldots, n_j$ in family $j = 1, \ldots, J$ such that

$$Y_{ij} \sim \text{Bernoulli}(y_{ij} \mid \pi_{ij}),$$

and

$$\pi_{ij} \equiv \Pr(Y_{ij} = 1 \mid \gamma_j) = \frac{1}{1 + \exp(-\beta_0 - X_{ij}\beta - Z_{ij}\gamma_j)}, \tag{2.12}$$

with the family-level random effect parameter vector γ_j distributed normally that has its expected value specified as a linear function of given covariates,

$$E(\gamma_j) = \theta_0 + W_{1j}\theta_1 + W_{2j}\theta_2 \tag{2.13}$$

and variance ϕ.

This model can be expressed with our notation as

```
formula <- list(pi = Y ~ X + tag(Z, "theta" | household),
           theta = ~ W1 + W2)
z.out <- zelig(formula, model = "logit.mixed",
           data = list(pi = indData, theta = houseData) )
```

where data for the π equation is loaded from the individual-level data set `indData`, and for the θ equation is loaded from a smaller, household-level dataset, `houseData`. The variable `household` is an index which links the individual level data and the household level data, and must be available in both datasets. Thus, the same model can also be expressed in reduced form, and loaded from a single dataset:

```
z.out <- zelig(Y ~ X + tag(Z, W1 + W2 | household),
               model = "logit.mixed", data = indData)
```

Note that the `household` strata must be identified even in the reduced form to index the observations in `W1` and `W2`.

3. A DEVELOPER'S INTERFACE

In addition to proposing an extension of the formula interface, we also provide some essential functions to transform this user-interface into data constructs useful for programmers. This section describes the computational infrastructure in Zelig that makes it easier to write new models (Section 3.1), makes those models compatible with the three-step Zelig framework for calculating quantities of interest (Section 3.3), and finally describes a self-generating GUI interface for the models included in Zelig (Section 3.4).

3.1 TOOLS FOR WRITING NEW MODELS

Consistent with the existing R framework, we implement a `multiple` class extension to the existing `model.matrix()` and `model.frame()` generic functions, in the form of `model.matrix.multiple()` and `model.frame.multiple()`. To ensure dispatch to the appropriate methods, we have also developed a suite of functions that simplify the process of writing a new model.

This process is divided into several steps that apply to every statistical model and estimator. First, the developer must write down the model, complete with all the parameters and the dimensionality of each parameter. Next, the developer needs some symbolic representation for the parameters (for example, we will use the syntax proposed in Section 2.3), and will transform that user-interface into vectors, matrices, or arrays that the developer can manipulate to produce estimates from the model.

The developer tools that we propose define the inputs to the statistical model or procedure in a function called `parse.formula()`, which has output of class `multiple` and ensures dispatch to the appropriate `model.frame()` and `model.matrix()` methods. There are two methods for defining models inputs. First, the developer can use some combination of the following arguments to define the model parameters: `req`, for parameters that correspond to dependent variables, without which the model cannot be fitted; `opt`, for parameters that do not correspond to dependent variables, which are optional, and which default to scalars if not specified by the user; and `ancil`, for scalar ancillary variables that do not vary over observations. Alternatively, the developer can drop these additional arguments and instead write a `describe.mymodel()` function to specify the model parameters (see Section 3.4), and invoke it using `parse.formula(formula, model = "mymodel")`. The first syntax is faster; the second allows more detailed descriptions and also serves the purpose of describing your model for our automated GUI creation facility. (The examples we give in Figures 2 and 3 show both options for illustrative purposes.)

Next, we offer tools to fix what is in our experience one of the most common programming mistakes in writing functions to be optimized. Procedures like `optim()` in R (and `maxlik` in Gauss or the optimization toolbox in Matlab) require a function to be optimized over a *single* parameter vector. This means that developers have to create starting values by concatenating unrelated parameters into one vector, and then they must figure out how to subset this vector into its constituent components inside the log-likelihood function. It typically leads to code that is either highly specific to the problem at hand,

```
beta <- par[1:4]; gamma <- par[5:6]; sigma <- par[7]
```

or general but extremely awkward and error prone, with constructs like:

```
beta <- par[(ncol(X)+ncol(Z)+2):
        (ncol(X)+ncol(Z)+2+ncol(W)+3)]
```

Our syntax for parsing model inputs makes possible an elegant and less error-prone approach by using functions that extract components of the parameter vector by name.

The syntax described here works with arbitrarily complicated models, with parameter vectors of any size and in any order, and with any number of equations, as long as each

set of parameters has been appropriately identified in the `parse.par()` statement. To illustrate this procedure, we begin with the familiar Gaussian normal regression model, with log-likelihood function:

$$\ln \mathrm{L}\,(\beta, \sigma^2 \mid y) = c - \frac{1}{2}\left\{ n \ln \sigma^2 + \frac{\sum_{i=1}^{n} (y_i - x_i \beta)^2}{\sigma^2} \right\}, \tag{3.1}$$

where $c = -\frac{n}{2} \ln 2\pi$. Obviously, the maximum of this function has a simple analytical solution, but for clarity, we write here the function necessary to optimize it numerically. The log-likelihood includes a parameter vector β and scalar σ^2, but in the function both must be stacked together as vec(β, σ^2) and then extracted separately. Consider the complete example in the `normal.regression` demo, and the excerpt presented in Figure 2.

The bracketed numbers in Figure 2 correspond to the following comments:

```
normal.regression <- function(formula, data, start.val = NULL, ...) {
  # fml <- parse.formula(formula, req = "mu", ancil = "sigma2") # [1a]
  fml <- parse.formula(formula, model = "normal.regression")    # [1b]
  D <- model.frame(fml, data = data)
  X <- model.matrix(fml, data = D)
  Y <- model.response(D)
  terms <- attr(D, "terms")

  start.val <- set.start(start.val, terms)                      # [2]

  ll.normal <- function(par, X, Y, n, terms) {                  # [3]
    beta <- parse.par(par, terms, eqn = "mu")                   # [3a]
    gamma <- parse.par(par, terms, eqn = "sigma2")              # [3b]
    sigma2 <- exp(gamma)
    -0.5 * (n * log(sigma2) + sum((Y - X %*% beta)^2 / sigma2))
  }

  res <- optim(start.val, ll.normal, method = "BFGS",           # [4]
               hessian = TRUE, control = list(fnscale = -1),
               X = X, Y = Y, n = nrow(X), terms = terms, ...)

  fit <- model.end(res, D)                                      # [5]
  class(fit) <- "normal"
  fit
}
```

Figure 2. Normal regression example using Zelig optimization tools. (Excerpts from the `normal.regression` demo.)

1. The `parse.formula(formula, ...)` function takes the user-specified formula and some information about the parameters in the statistical model (provided by the developer). There are two ways to use `parse.formula()`:

   (a) As in line (1a), we use the `req`, `opt`, and `ancil` options. In the case of normal regression, the parameter vector β corresponds to $E(Y_i) = \mu_i = X_i\beta$ and is hence a required user input, but the ancillary parameter σ^2 is always estimated as a scalar.

   (b) Write a `describe.mymodel()` function (see 3.4) to work with the GUI. If you had written a function called `describe.normal.regression()`, you could use the code in line (1b) instead of (1a), as in the example above, to avoid defining the model's parameters in multiple locations.

2. Automatically set starting values using the parameters identified in `parse.formula()`, and subsequently stored in `terms`. If any equations are constrained, you may replace the default values (zero for all parameters), by using `put.start(start.val, eqn, value)`, which works for either scalar or vector parameters.

3. The log-likelihood function corresponds to the mathematical expression above. Within the log-likelihood, use the `parse.par()` function to extract different sets of parameters from the vector to be optimized, `par`. Line (3a) extracts the parameters which correspond to the mean equation μ_i, and line (3b) extracts the scalar parameter that corresponds to σ^2, which is reparameterized in the subsequent line to satisfy the constraint $\sigma^2 > 0$.

4. The call to the optimization routine, `optim()`.

5. The tidying function `model.end()` takes the optimized output and codes some additional meta-data so that `model.frame()` and `model.matrix()` will work in subsequent steps.

3.2 MANAGING PARAMETERS IN MODELS WITH MORE THAN ONE DEPENDENT VARIABLE

Many statistical methods relate explanatory variables x_i to a dependent variable of interest y_i for each observation ($i = 1, \ldots, n$) through a possibly nonlinear function of a linear predictor η_i. Let β be a set of parameters that correspond to each column in X, which is an $n \times k$ matrix with rows x_i. For a single equation model, the linear predictor is

$$\eta_i = x_i\beta, \tag{3.2}$$

where η is the set of η_i (for $i = 1, \ldots, n$) and is usually represented as an $n \times 1$ matrix.

For a two-equation model, the linear predictor becomes a matrix with two columns where each row is given by

$$\eta_i = (\eta_{i1}, \eta_{i2}) = (x_{i1}\beta_1, x_{i2}\beta_2). \tag{3.3}$$

With η as an $n \times 2$ matrix, we now have four choices for constructing the linear predictor: An *equation-by-equation* layout, which pulls out the X matrix for one equation at a time; an *intuitive* layout, which stacks matrices of explanatory variables, and provides an easy visual representation of the relationship between explanatory variables and coefficients; a *memory-saving* layout, which reduces the overall size of the X and β matrices; and a *computationally efficient* layout, which takes advantage of vectorization for speed.

Each of these methods is associated most closely with *vector*, *stacked vector*, or *matrix* representations of the coefficients. The choice of the format for explanatory variables and parameters also determines how easy it is to allow constraints across equations.

For a running example in describing the different methods, we use this three-equation system from the bivariate probit model, with

```
formulae <- list(mu1 = y1 ~ tag(x1, "gamma") + x2,
                 mu2 = y2 ~ tag(x2, "gamma") + x3,
                 rho = ~ x4)
```

and with a constraint of equality on the first coefficient across the first two equations. For simplicity, we ignore the ρ equation below since parameters cannot be constrained between μ and ρ. The above example may be examined in detail in the `bivariate.probit` demo included in Zelig.

The Equation-by-Equation Layout. Choosing `model.matrix(..., eqn = "mu2")` outputs the X matrix corresponding only to the equation for μ_{i2}. It produces an $n \times 3$ matrix for X (the two variables and a constant term) and thus directly generalizes the standard `model.matrix()` in a simple and easy-to-understand way. To extract parameters in a format convenient for this X matrix representation, we would use the vector representation of the parameters, which we do by using `parse.par(..., shape="vector", eqn="mu2")` (where `shape="vector"` could be omitted since it is the default when extracting parameters from a single equation).

Unfortunately, implementing constraints across equations would be difficult with this method for any particular example, and would require tedious coding to make it work in general. Although this method is the first one that most think of when they desire multiple-equation generalizations, they quickly learn that something even more general and sophisticated is required. In particular, multiple equations need to be treated as a set rather than entirely separately. The following three methods do exactly this.

The Intuitive Layout. A stacked matrix of X and stacked vector β is probably the most *visually* intuitive configuration. Let $J = 2$ be the number of equations in the bivariate probit model, n be the number of observations, and υ be the number of unique covariates across both equations. Then, `model.matrix(..., shape = "stacked")` yields a $(Jn \times \upsilon)$ matrix of explanatory variables. For the example above, we have:

$$X = \begin{pmatrix} 1 & 0 & x_1 & x_2 & 0 \\ 0 & 1 & x_2 & 0 & x_3 \end{pmatrix}, \tag{3.4}$$

where x_3 from the first equation and x_4 from the second are in the same column because their (tagged) coefficients are constrained to be equal. Correspondingly, we extract β as a

stacked vector, using `parse.par(..., shape="vector")`, producing

$$(\beta_0^{\mu_1}\ \beta_0^{\mu_2}\ \beta_\gamma\ \beta_{x_2}^{\mu_1}\ \beta_{x_3}^{\mu_2})', \tag{3.5}$$

where $\beta_0^{\mu_1}$ and $\beta_0^{\mu_2}$ are the intercept terms for Equations (2.1) and (2.2), respectively. Since X is $(2n \times 5)$ and β is (5×1), the matrix product of the two is the stacked $(2n \times 1)$ linear predictor η. Although difficult to manipulate (since observations are indexed by i and $2i$ for each $i = 1, \ldots, n$), it is easy to see that we have turned the two equations into one large X matrix and one long vector β, which is analogous to the familiar single-equation η.

The Memory-Efficient Layout. Choosing a "compact" X matrix and matrix β is usually the most memory-efficient configuration: `model.matrix(..., shape = "compact")` produces an $n \times v$ matrix, where v is the number of unique variables in all of the equations (4 in this case, since the intercept term is identically valued in both equations, and so counts only as one variable, and the the *unique* variables are x_1, x_2, and x_3). Let x_1 be an $n \times 1$ vector representing variable `x1`, x_2 be `x2`, and so forth. This leaves:

$$X = (1\ x_1\ x_2\ x_3) \quad \beta' = \begin{pmatrix} \beta_0^{\mu_1} & \beta_\gamma & \beta_{x_2}^{\mu_1} & 0 \\ \beta_0^{\mu_2} & 0 & \beta_\gamma & \beta_{x_3}^{\mu_2} \end{pmatrix}, \tag{3.6}$$

where the β matrix is constructed via `parse.par(..., shape="matrix")`. β_{land} is used twice to implement the constraint, and the number of empty cells is minimized by implementing the constraints in β rather than X. Furthermore, since X is $(n \times 4)$ and β is (4×2), $X\beta = \eta$ is $n \times 2$.

The Computationally Efficient Layout. Choosing array X and vector β is probably the the most computationally efficient configuration: `model.matrix(..., shape = "array")` produces an $n \times k \times J$ array where J is the total number of equations and k is the number of unique parameters across all the equations. Denote the number of parameters in equation j as k_j. Then, since some parameter values may be constrained across equations, $k \leq \sum_{j=1}^{J} k_j$. If a variable is not in a certain equation, it is observed as a vector of zeros. With this option, *each* x_i matrix becomes:

$$\begin{pmatrix} 1 & 0 & x_{i1} & x_{i2} & 0 \\ 0 & 1 & x_{i2} & 0 & x_{i3} \end{pmatrix}. \tag{3.7}$$

By stacking each of these x_i matrices along the first dimension, we get X as an array with dimensions $n \times k \times J$. Correspondingly, β is a stacked vector, created with `parse.par(..., shape="vector")`, and having elements

$$(\beta_0^{\mu_1}\ \beta_0^{\mu_2}\ \beta_\gamma\ \beta_{x_2}^{\mu_1}\ \beta_{x_3}^{\mu_2})'. \tag{3.8}$$

To multiply the X array with dimensions $(n \times 5 \times 2)$ and the (5×1) β vector, we *vectorize* over equations as follows:

```
eta <- apply(X, 3, '%*%', beta)
```

The linear predictor η is therefore a $(n \times 2)$ matrix.

3.2.1 Illustrations

To illustrate how easy it is to interchange these options, we introduce a concrete example in the `bivariate.probit` demo in Zelig. Consider the bivariate probit example introduced in Equations (2.2)–(2.4). We begin with the memory-efficient layout in Figure 3 and show that we only need to modify a few lines of code to change from one of these schemes to another.

To change to the intuitive option or the computationally efficient option, we change only a few lines of code. For the intuitive option, at Comment (2), we switch to the option in line (2b)

```
X <- model.matrix(fml, data = D, shape = "stacked",
     eqn = c("mu1", "mu2"))
```

and at Comment (3), to option (3b)

```
Beta <- parse.par(par, terms, shape = "vector",
        eqn = c("mu1", "mu2"))
```

and at Comment (4), to option (4b)

```
mu <- X %*% Beta; mu <- matrix(mu, ncol = 2)
```

To switch to the computationally efficient layout, replace the line at Comment (2) with line (2c)

```
X <- model.matrix(fml, data = D, shape = "array",
     eqn = c("mu1", "mu2"))
```

and at Comment (3) with line (3c)

```
Beta <- parse.par(par, terms, shape = "vector",
        eqn = c("mu1", "mu2"))
```

and at Comment (4) with line (4c)

```
mu <- apply(X, 3, '%*%', Beta)
```

Even if your optimizer calls directly a C or FORTRAN routine using functions such as `.C()` and `.Fortran()`, one can use combinations of Zelig's `model.*()` and `parse.par()` functions to set up the data structures needed to obtain the linear predictor (or the model's equivalent) before passing these data structures to the estimation routine.

3.3 WRAPPING EXISTING PACKAGES

Although Zelig offers some tools for those writing new R packages in Section 3, developers need not use these tools to incorporate existing code into Zelig. This section assumes that you have some model already coded up and would like to incorporate it into the Zelig framework for estimating quantities of interest without modifying the original code. To accomplish this, we use the computational framework illustrated in Figure 4.

```
bivariate.probit <- function(formula, data, start.val = NULL, ...) {
  # fml <- parse.formula(formula, req=c("mu1","mu2"), opt="rho") # [1a]
  fml <- parse.formula(formula, model = "bivariate.probit")      # [1b]
  D <- model.frame(fml, data = data)

  X <- model.matrix(fml, data = D, eqn = c("mu1", "mu2")) # [2a]
  # X <- model.matrix(fml, data = D, shape = "stacked",   # [2b]
  #                 eqn = c("mu1", "mu2"))
  # X <- model.matrix(fml, data = D, shape = "array",     # [2c]
  #                   eqn = c("mu1", "mu2"))

  Xrho <- model.matrix(fml, data = D, eqn = "rho")
  Y <- model.response(D)
  terms <- attr(D,"terms")
  start.val <- set.start(start.val, terms)
  start.val <- put.start(start.val, 1, terms, eqn = "rho")

  log.lik <- function(par, X, Y, terms) {

    Beta <- parse.par(par, terms, eqn = c("mu1", "mu2"))   # [3a]
    # Beta <- parse.par(par, terms, shape = "vector",      # [3b] & [3c]
    #                   eqn = c("mu1", "mu2"))

    gamm <- parse.par(par, terms, eqn = "rho")
    rho <- (exp(Xrho %*% gamm) - 1) / (1 + exp(Xrho %*% gamm))

    mu <- X %*% Beta                                       # [4a]
    # mu <- X %*% Beta; mu <- matrix(mu, ncol = 2)         # [4b]
    # mu <- apply(X, 3, '%*%', Beta)                       # [4c]
       [... main log-likelihood calculation, omitted ...]
    return(llik)
  }
  res <- optim(start.val, log.lik, method = "BFGS",
               hessian = TRUE, control = list(fnscale = -1),
               X = X, Y = Y, terms = terms, ...)
  fit <- model.end(res, D)
  class(fit) <- "bivariate.probit"
  fit
}
```

Figure 3. Bivariate probit function, from the `bivariate.probit` demo. To use the memory-efficient default, use lines 2a, 3a, and 4a; for the intuitive option, choose lines 2b, 3b, and 4b, and for the computationally efficient option, lines 2c, 3c, and 4c. At Comments (2) and (3), you may optionally simplify `eqn = c("mu1", "mu2")` to `eqn = "mu"`.

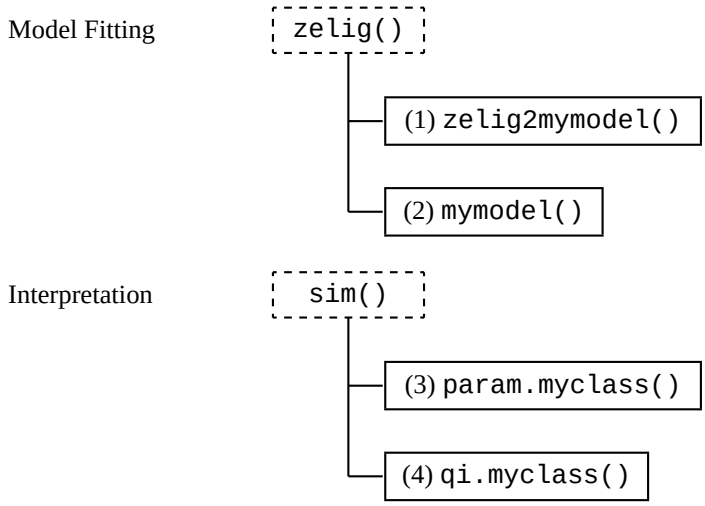

Figure 4. A developer's framework for Zelig, where `mymodel` is both the name of the model and the function that fits the model, and `myclass` is the class of the output from the statistical procedure, for which appropriate `print()` and `summary()` methods have been defined.

The Zelig framework works by taking advantage of R's lazy evaluation, and the flexibility of R classes. The `zelig()` function itself works by redefining the call to `zelig(..., model="mymodel")` to a call to `mymodel()`, via a developer-supplied wrapper function called `zelig2mymodel()`. Since this dispatch method does not rely on classes, `zelig()` can wrap any statistical procedure by evaluating first `zelig2mymodel()` to redefine the call, then the new call. The output from `zelig()` retains the original class (whether S3 or S4), such that any pre-existing generic functions with defined methods will work. (If the original output is an S3 object, `zelig()` simply adds a few elements necessary for subsequent parts of our framework. If the original output is an S4 object, we create a new object that inherits attributes from the original, and adds slots for the new elements.) The simulation procedure relies on two generic functions: one to simulate parameters, `param.myclass()`; and one to calculate quantities of interest appropriate to the model, `qi.myclass()`. Since the output from `sim()` is standardized, we have written a `summary()` method for this output.

Thus, a developer only needs to write three functions (in addition to the functions that fit the model) to take advantage of Zelig's interpretative framework:

1. `zelig2mymodel()` to redefine the call;
2. `param.myclass()` to simulate parameters; and
3. `qi.myclass()` to calculate model-specific quantities of interest.

Because the initial dispatch from `zelig()` relies on the wrapper, which is generated by specifying `zelig(..., model = "mymodel")`, models can be added to this framework without any modifying the `zelig()` function itself.

3.4 A DYNAMICALLY GENERATED GUI

The power and flexibility of R as a statistical programming language and computational environment have made graphical user interfaces (GUIs) difficult to implement. The general advantage of a GUI is that it lets analysts take advantage of the work of R developers without having to learn R's command-line interface that is as powerful for some as it is intimidating for others. A good GUI would therefore make the work of R developers accessible to a much larger audience than present.

Although many attempts have been made at a GUI, we introduce here the concept of a *dynamic* or self-generating GUI, which can be extended without modifying the functions which govern the GUI itself. Thus, there is no static database of functions, arguments, and types of inputs accepted as arguments. Rather, the GUI queries Zelig about the included models, and the types of accepted inputs, and uses this information to render the GUI. Since developers who use our model-writing tools described in Section 3.1 have already defined the parameters required for their model, it is a relatively simple step to transform this information into a data structure that can be queried by other programs.

For any model `mymodel`, developers can create a function called `describe.mymodel()`, which takes no arguments and returns a list with standard elements describing the model. This list includes at a minimum the `category` into which the model falls using a standard set of categories we have developed based on the type of dependent variable or variables that are allowed (continuous unbounded, continuous bounded, dichotomous, ordinal, multinomial, count, and mixed), and a list of sets of equation-level `parameters` (in the linear normal regression model in Equation (2.1), this includes μ and σ). Each element of the `parameters` list includes the minimum and maximum number of `equations` allowed, whether tags are allowed (`tagsAllowed`), whether the parameter should be associated with a dependent variable (`depVar`) and explanatory variables (`expVar`). For the bivariate probit model summarized in Equations (2.2)–(2.4), this function is displayed in Figure 5. (Other information which enable additional functionality are documented in the Zelig manual and may be included as well.)

Writing this simple summary of the model parameters makes a model accessible to a GUI. To build the initial model selection menu, the GUI merely needs to know how many `describe.*()` functions exist in attached environments, to collect the information stored in those functions, and then render it as a series of predefined fields in the GUI. Since the allowed types for the explanatory and dependent variables can also be coded into the `describe.*()` function, this gives the GUI sufficient information to check the variables input into different fields, to ensure that they satisfy the model's assumptions. For an example of a Zelig-based GUI, visit one of the installations of the Dataverse Network such as *http://dvn.iq.harvard.edu/dvn* (see also King 2007 or the project Web site at *http://TheData.org*).

Using this approach for describing model parameters also simplifies writing a new model, for those who use the developer tools described in Section 3.1. Rather than defining the model parameters using the ad hoc `req`, `opt`, and `ancil` options, developers can use instead `parse.par(formula, model="mymodel")`, assuming that `describe.mymodel()` exists. Thus, developers might choose to use lines (1b) in Figures 2 and 3, rather than the less-precise definitions in lines (1a).

```
describe.bivariate.probit <- function() {
  category <- "dichotomous"
  mu <- list(equations = 2,          # 2 parameters
             tagsAllowed = TRUE,     # for the mean
             depVar = TRUE,
             expVar = TRUE),
  rho <- list(equations = 1,         # 1 parameter
             tagsAllowed = FALSE,    # for the correlation
             depVar = FALSE,
             expVar = TRUE),
  pars <- parameters(mu = mu, rho = rho)
  list(category = category, parameters = pars)
}
```

Figure 5. The relevant `describe.mymodel()` function for the `"bivariate.probit"` model.

4. CONCLUDING REMARKS

The original developers of the S and R projects "wanted users to be able to begin in an interactive environment, where they did not consciously think of themselves as programming. Then as their needs became clearer and their sophistication increased, they should be able to slide gradually into programming, when the language and system aspects would become more important" (*http://stat.bell-labs.com/S/history.html*). Much of the work since the early days has focused on the extreme ends of this "slide"— at one end shoring up the foundations of the basic statistical computing language, most recently with sophisticated tools like methods and classes, and at the other end the development of numerous separate packages by many independent investigators.

Our intention here has been to help shore up the middle ground, to abstract features of a large fraction of statistical approaches, in particular a far wider range than is covered by the standard R single-equation framework, and to bring some unity to the diversity of statistical approaches and syntaxes, all without changing any of the existing packages, or their approaches, notation, or examples. We also feel that it is time to recognize that most users will never "slide gradually into programming." Instead, we need to develop common tools so that programmers slide gradually into into producing methods that applied researchers can use directly. Although the developments offered here will not come close to unifying the diverse methodological subfields of different substantive disciplines, we hope that it facilitates the dispersion of methodological advances across fields through the use and development of an increasingly powerful and diverse set of statistical approaches accessible to all.

ACKNOWLEDGMENTS

Our thanks to the many contributors to and users of the Zelig project; to Jeff Gill, Uwe Ligges, Brian Ripley, anonymous referees, the associate editor, and the editor for helpful comments; and to the National Institutes of Aging (P01 AG17625-01), the National Science Foundation (SES-0318275, IIS-9874747, SES-0550873), the Mexican Ministry of Health, and the Princeton University Committee on Research in the Humanities and Social Sciences for research support.

[Received November 2006. Revised April 2008.]

REFERENCES

Becker, Richard A., Chambers, John M., and Wilks, Allan R. (1988), *The New S Language*, New York: Wadsworth.

Bishop, Christopher M. (1995), *Neural Networks for Pattern Recognition*, Oxford: Oxford University Press.

Ho, Daniel E., Imai, Kosuke, King, Gary, and Stuart, Elizabeth A. (Forthcoming), "MatchIt: Nonparametric Preprocessing for Parametric Causal Inference," *Journal of Statistical Software*. Available online at http://gking.harvard.edu/matchit.

——— (2007), "Matching as Nonparametric Preprocessing for Reducing Model Dependence in Parametric Causal Inference," *Political Analysis*, 15, 199–236. *http://gking.harvard.edu/files/abs/matchp-abs.shtml*.

Honaker, James, and King, Gary (2007), "What to do About Missing Values in Time Series Cross-Section Data," available online at *http://gking.harvard.edu/files/abs/pr-abs.shtml*.

Honaker, James, King, Gary, and Blackwell, Matthew (2006), "Amelia II: A Program for Missing Data." Available online at *http://gking.harvard.edu/amelia*.

Ihaka, Ross, and Gentleman, Robert (1996), "R: A Language for Data Analysis and Graphics," *Journal of Computational and Graphical Statistics*, 5, 299–314.

Imai, Kosuke, King, Gary, and Lau, Olivia (2006), "Zelig: Everyone's Statistical Software," available online at *http://gking.harvard.edu/zelig*.

Imbens, Guido W. (2004), "Nonparametric Estimation of Average Treatment Effects Under Exogeneity: A Review," *Review of Economics and Statistics*, 86, 4–29.

King, Gary (1989), *Unifying Political Methodology: The Likelihood Theory of Statistical Inference*, Ann Arbor: Michigan University Press.

——— (1995), "Replication, Replication," *PS: Political Science and Politics* 28 (September): 443–499. Online at *http://gking.harvard.edu/files/abs/replication-abs.shtml*.

——— (2007), "An Introduction to the Dataverse Network as an Infrastructure for Data Sharing," *Sociological Methods and Research*, 36 (2), 173–199. Available online at *http://gking.harvard.edu/files/abs/dvn-abs.shtml*.

King, Gary, Murray, Christopher J.L., Salomon, Joshua A., and Tandon, Ajay (2004), "Enhancing the Validity and Cross-cultural Comparability of Measurement in Survey Research," *American Political Science Review*, 98 (February), 191–207. Available online at *http://gking.harvard.edu/files/abs/vign-abs.shtml*.

King, Gary, Honaker, James, Joseph, Anne, and Scheve, Kenneth (2001), "Analyzing Incomplete Political Science Data: An Alternative Algorithm for Multiple Imputation," *American Political Science Review*, 95 (March): 49–69. Available online at *http://gking.harvard.edu/files/abs/evil-abs.shtml*.

King, Gary, and Zeng, Langche (2006), "The Dangers of Extreme Counterfactuals," *Political Analysis*, 14 (2), 131–159. Available online at *http://gking.harvard.edu/files/abs/counterft-abs.shtml*.

——— (2007), "When Can History Be Our Guide? The Pitfalls of Counterfactual Inference," *International Studies Quarterly* (March), 183–210. Available online at *http://gking.harvard.edu/files/abs/counterf-abs.shtml*.

King, Gary, Tomz, Michael, and Wittenberg, Jason (2000), "Making the Most of Statistical Analyses: Improving Interpretation and Presentation," *American Journal of Political Science*, 44 (April), 341–355. Available online at *http://gking.harvard.edu/files/abs/making-abs.shtml*.

R Development Core Team (2008), *R: A Language and Environment for Statistical Computing*, Vienna, Austria: R Foundation for Statistical Computing. *http://www.R-project.org*

Rubin, Donald B. (1973), "Matching to Remove Bias in Observational Studies," *Biometrics*, 29, 159–184.

——— (1987), *Multiple Imputation for Nonresponse in Surveys*, New York: Wiley.

Stoll, Heather, King, Gary, and Zeng, Langchee (2005), "WhatIf: Software for Evaluating Counterfactuals," *Journal of Statistical Software*, 15 (4). Available online at *http://www.jstatsoft.org/index.php?vol=15*.